Composition in Java

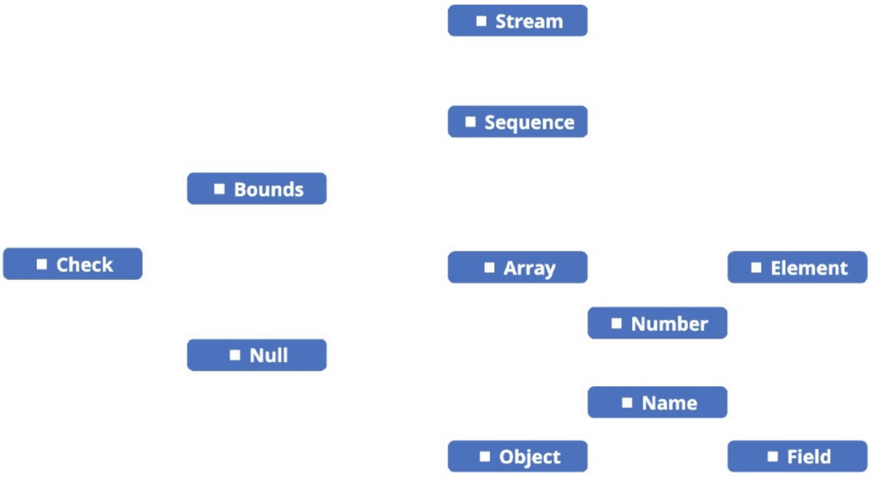
Workbook **14**

Null, Arrays, Streams

Student name:	TA signature:

Based on Photo by Philippe Murray-Pietsch on Unsplash

Concepts Check off understood concepts, connect related concepts, label connections



■ Stream

■ Sequence

■ Bounds

■ Check

■ Array

■ Element

■ Number

■ Null

■ Name

■ Object

■ Field

Make sure you can **explain** each concept and each connection, you can provide **examples**, and you can **identify** them in a given piece of code.

Names Circle the methods, underline the types

```
java.util.ArrayList, java.util.stream.Stream
```

Null – Tony Hoare’s Billion Dollar Mistake

Turing Award winner **Tony Hoare*** introduced null references in ALGOL W in 1965 “simply because it was so easy to implement”. In a 2009 presentation** he talks about that decision and considers it his “**billion-dollar mistake**”.

References (arrows in memory diagrams) can be implemented as addresses. Hoare decided that the address 0 (the “null reference”) had a special meaning: it means that the reference does **not** refer to any object.

```
Person myFriend = new Person();
Person yourFriend = null;
```

Programming languages like Java provided a special literal to denote null references: `null`.

Draw the memory diagram (stack, heap) of the state at the end of the above code:

Assume we now execute the following code:

```
public static String announceGuest(Person guest) {
  return "Let me introduce " + guest.getName();
}
```

What could happen when we call that method?

Rewrite the `announceGuest` using `Option`, avoiding the billion dollar mistake:

```
public static
  return
}
```

Default Values of Reference Types

In Java, instance variables are automatically initialized to the **default** value of their type: an `int` to `0`, a `double` to `0.0`, a `boolean` to `false`, a reference to `null`.

Why? The default value of a type is the value of that type that is represented using all zero bits. This allows a cheap way to initialize fields of newly allocated objects to their default values: before objects are allocated, all the bits that make up an object are reset to zero at once. Thus, when allocating an object, there is no need for code to explicitly set instance variables to any value, because the memory in which all of those variables exist will have been zeroed out. Of course, if you do want to initialize an instance variable, you can still do that; but you don’t have to.

* https://amturing.acm.org/award_winners/hoare_4622167.cfm
** https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

Arrays – Low-Level Fixed-Length "Lists"

Besides various convenient collection classes, Java also provides a low-level way to store **fixed**-length lists: **arrays**. An array, like an object, is created on the **heap**. And like an object, it **cannot** shrink and grow. An **object** has a fixed set of **fields**, an **array** has a fixed set of **elements**. Fields are **named** memory locations inside the object. Elements are **numbered** memory locations inside the array.

While the type of a list is specified as `List<T>`, for a given element type `T`, the type of an array is specified as `T[]`. Unlike for generic type parameters (like the `T` in `List<T>`), the element type of an array can be any type, even a primitive type.

Draw a memory diagram (heap) showing the result of each expression. The expression in brackets determines the **size** (number of elements) of the array:

`new Pos(1, 2)`	`new int[3]`	`new String[2]`

When allocating an array like above, its elements get initialized to the default value of the element type (e.g., `0` for `int`, `null` for `String`). You can allocate **and initialize** an array in one expression as follows:

```
new int[] {1, 2}                        new String[] {"Hi", "Ho", "Go"}
```

To **read** the value from an array element, you can write ARRAY[INDEX]. Instead of ARRAY, write an expression that evaluates to an array, and instead of INDEX, write an expression of type `int` that evaluates to a value greater or equal to zero.

To **write** the value of an array element, you can write ARRAY[INDEX] on the left hand side of an assignment operator: `words[5] = "ciao"`.

To determine the **length** of an array, you can access its immutable `length` field.

Here is a comparison of a Java `List` (e.g., `ArrayList`) and a Java array:

	`ArrayList` class	**Array**
Type	`ArrayList<T>`	`T[]`
Variable Declaration	`ArrayList<String> o;`	`String[] a;`
Allocate	`new ArrayList<String>()`	`new String[5]`
Length	`o.size()`	`a.length`
Read element	`o.get(4)`	`a[4]`
Write element	`o.set(1, "One")`	`a[1] = "One"`
Append	`o.add("More")`	*not possible*
Insert	`o.add(2, "between")`	*not possible*
Remove	`o.remove(2)`	*not possible*

What is the value of the following expressions? Briefly explain.

`(new int[2])[1 - 1]`		`new int[3 + 2].length`	
`new int[] {1, 2}[1]`		`new String[0].length`	

Using Arrays to Implement our own ArrayList Class

If Java did not provide an `ArrayList` class, we probably would want to write our own. The main benefit of using an `ArrayList` over using a plain array is that `ArrayLists` can **grow**, while arrays have a **fixed** size.

Complete the following implementation. The `size` field keeps track of how many elements we have in our list. The current capacity (length of the `elements` array) can be bigger than the number of elements we currently contain. If we use all the elements of the array, that is, if `size == elements.length`, and we need to add a new element, we have to "grow" the list. We do that in the `grow` method by allocating a new, bigger array, and by copying (you may want to use a `for`-loop) the element values from the old array to the new array.

```
public class MyStringArrayList {

  private int size;
  private String[] elements;

  public MyStringArrayList(int capacity) {
    assert capacity >= 0;
    elements = new String[capacity];
    size = 0;
  }
  public int size() {
    return size;
  }
  public String get(int index) {
    return elements[index];
  }
  public void set(int index, String value) {
    elements[index] = value;
  }
  public void add(String value) {
    if (size >= elements.length) {
      grow();
    }
    elements[size] = value;
    size++;
  }
  private void grow() {





  }
}
```

You had to decide how large the new array should be. You could make it just one element longer. That would work, but performance would not be optimal (you would copy the entire array for each newly appended element). A better strategy is to double the size whenever you need to grow.

Sorting an Array

Tony Hoare didn't just invent **`null`**, but he also invented something much more beneficial: **quicksort**!

Here is a quick sort implementation similar to the one in Workbook 8. This uses a recursive data structure (a `Sequence`). The function is **pure** and the data structure is **immutable**.

```
public static <T> Sequence<T> sort(
  Function2<T,T,Boolean> lessEqual,
  Sequence<T> values
) {
  if (isEmpty(values)) {
    return empty();
  } else {
    T pivot = first(values);
    Sequence<T> smaller = filter(x ->  lessEqual.apply(x, pivot), rest(values));
    Sequence<T> greater = filter(x -> !lessEqual.apply(x, pivot), rest(values));
    return concat(
        sort(lessEqual, smaller),
        cons(
          pivot,
          sort(lessEqual, greater)
        )
    );
}
```

Let's write an implementation **using arrays**. The function does not return any value; its computation happens "in place", it has the side-effect of mutating the array that is passed in as an argument. The function is impure (it modifies variables on the heap) and the data structure (the array) is mutable.

```
public static <T> void sort(Function2<T,T,Boolean> lessEqual, T[] values) {
  return sort(lessEqual, values, 0, values.length - 1);
}
public static <T> void sort(Function2<T,T,Boolean> lessEqual, T[] values,
  int low, int high
) {
  if (low >= high || low < 0) { return; }
  int pivotIndex = partition(lessEqual, values, low, high);
  sort(lessEqual, values, low, pivotIndex - 1);
  sort(lessEqual, values, pivotIndex + 1, high);
}
public static <T> int partition(Function2<T,T,Boolean> lessEqual, T[] values,
  int low, int high
) {
  T pivot = values[high];
  int i = low;
  for (int j = low; j < high; j++) {
    if (lessEqual.apply(values[j], pivot)) {
      swap(values, i, j);
      i = i + 1;
    }
  }
  swap(values, i, high);
  return i;
}
```

Implement the `swap` method that is used by the `partition` method above:

```
public static



}
```

What is the difference between the sequence-based implementation and the array-based implementation in terms of pivot selection?

Reversing an Array

Implement a method that gets an array and returns a new array that contains the elements of the original array in reverse order. Use a `for`-loop with index:

```
public static String[] reverse(String[] original) {


}
```

Implement a method that gets an array and reverses its contents in-place. Use a `for`-loop with index:

```
public static void reverse(String[] original) {



}
```

Searching an Array

Complete this method that performs a **binary search** in an array (it assumes that the array is sorted):

```
public static Option<Integer> binarySearch(String[] values, String key) {
  int low = 0;
  int high = values.length - 1;
  while (low <= high) {
    int mid = (high + low) / 2;
    int comparison = values[mid].compareTo(key);
    if (comparison < 0) {
      low = mid + 1;
    } else if (comparison > 0) {
      high = mid – 1;
    } else {
      return
    }
  }
  return
}
```

"Two-Dimensional Arrays" – Arrays of Arrays

How does one represent a two-dimensional structure, such as a tile-based game board (e.g., the Maze for pacman) or a matrix?

You could use a sequence of sequences of elements:

```
Sequence<Sequence<Tile>> board = of(
  of(WALL, FLOOR, WALL),
  of(WALL, FLOOR, FLOOR)
);
```

You can do the same by creating an array of arrays of elements:

```
Tile[][] board = {
  { WALL, FLOOR, WALL },
  { WALL, FLOOR, FLOOR }
};
```

Draw a memory diagram (stack and heap) of the above array-based board:

Implement a method that reduces the board into a multi-line string (use `"\n"` to terminate each line), looking like this:

```
X.X
X..
```

Use **`for`-loops** (with indices named `row` and `col`):

```
public static String renderToString(Tile[][] maze) {










}
```

“Multi-Dimensional Arrays” – Arrays of Arrays of …

In Java you can create arrays with any number of dimensions. More precisely, you really can only create one-dimensional arrays, but you can put references to arrays inside those arrays, and that nesting allows you to create arbitrarily deeply nested array structures.

Draw a memory diagram (stack and heap) at the end of this method:

```
public static void demo() {
  int i;           // primitive int
  int[] vi;        // reference to array of int
  int[][] vvi;     // reference to array of arrays of int
  int[][][] vvvi;  // reference to array of arrays of arrays of int
  i = 5;
  vi = new int[3];
  vvi = new int[2][4];
  vvvi = new int[1][2][2];
  vvvi[0][1] = vvi[1];
  vvi[1][2] = 42;
  int x = vvvi[0][1][2];
}
```

As you can see in your diagram, there can be **sharing** when using arrays. This means there is **aliasing**: multiple ways to refer to the same memory location.

The diagram doesn’t show any **cycles**. Can you create cycles with only arrays?

Dereference Checks and Bounds Checks

Let's play with the following method:

```
public static int get(int[] values, int index) {
  return values[index];
}
```

What happens in the following situations?

Invocation	Return value	Exception
get(new int[3], 3)		
get(new int[0], 0)		
get(new int[3], -1)		
get(null, 0)		

In Java, every array access is **bounds-checked**. And every dereference operation (following a reference to an object or array) is **checked** against **null**. If an index is out of bounds, Java throws an IndexOutOfBounds exception. If a null reference is dereferenced, Java throws a NullPointerException. Such exceptions can terminate the program abruptly. You usually see a stack trace in the terminal in that case. But it's possible for programmers include code that catches the exceptions and continues with the program execution.

Lower-level, unsafe languages like C do not perform these safety-checks. Programs may crash uncontrollably (often with very little information on what went wrong) or, worse, they may have arbitrary effects and produce arbitrary results, often without anyone noticing the problem (until a plane crashes or rocket explodes).

For example, in C you can easily access an array element that does not exist (e.g., element at index 1000 in an array of length 3). The code generated by a C compiler will simply add 1000 to the start address of the array and read (or write!) whatever is at the resulting memory location. This kind of "buffer overflow" is a key cause of security bugs in today's software!

Avoiding Runtime Exceptions

Java's type system prevents the above kinds of problems. However, getting an exception at runtime in Java still can be painful. Especially if that leads to a production system going down. Thus, it is best to completely avoid problems by using approaches like the Option type we discussed. This lets the compiler force us to handle every possible problem (like you saw in labs 11 & 12).

For example, our binarySearch function returns an Option<Integer>, so it can tell us whether or not the value was found. If it was found it returns some(index), otherwise it returns none(). We get the Option and all we can do with it is fold it. fold requires arguments for **both** situations; so we can't ignore the problem case!

Complete the code to produce "Found at index: …" or "Not found."

```
binarySearch(friends, "Voldemort").fold(
```

Java Streams

A significant fraction of this course centered around the `Sequence` interface, and how to use `map`, `filter`, and `reduce`, to conveniently process sequences. We included this material for two reasons: (1) it connects to PF1's use of lists (sequences are equivalent to BSL lists), and (2) it connects to Java's more advanced **Streams API** (map, filter, and reduce on sequences is equivalent to—but simpler and cleaner than—the same operations on Java streams).

The Java API contains an entire package (`java.util.stream`) with classes that are similar to **sequences**. Those classes are called **streams**. The main way to process the elements in a stream are `map`, `filter`, and `reduce`.

Creation

The Java API provides various ways to create streams. Unlike with sequences, however, you don't create streams with `cons` or `of`. You can, however, create streams from an existing collection, like a `List`:

```
public static Stream<String> toStream(List<String> strings) {
  return strings.stream();
}
```

Mapping

Let's map strings to integers by parsing them:

```
public static Sequence<Integer> mapDemo(Sequence<String> strings) {
  return map(Integer::parseInt, strings);
}

public static Stream<Integer> mapDemo(Stream<String> strings) {
  return strings.map(Integer::parseInt);
}
```

Filtering

Let's keep only the non-empty strings:

```
public static Sequence<String> filterDemo(Sequence<String> strings) {
  return filter(s -> !s.isEmpty(), strings);
}

public static Stream<String> filterDemo(Stream<String> strings) {
  return strings.filter(s -> !s.isEmpty());
}
```

Reduction

Let's concatenate all the strings:

```
public static String reduceDemo(Sequence<String> strings) {
  return reduce("", String::concat, strings);
}

public static String reduceDemo(Stream<String> strings) {
  return strings.reduce("", String::concat);
}
```

Now that you **master sequences** you are ready to discover Java's streams and all the concurrency and performance goodies they provide!